Refactoring as Testability Transformation

Mark Harman
University College London, Department of Computer Science, CREST Centre,
Malet Place, London, WC1E 6BT, UK.

***Abstract*—This paper[1] briefly reviews the theory of Testability Transformation and outlines its implications for and relationship to refactoring for testing. The paper introduces testability refactorings, a subclass of Testability Transformations and discusses possible examples of testability refactorings. Several approaches to testability refactoring are also introduced. These include the novel concept of test–carrying code and the use of pareto optimization for balancing the competing needs of machine and human in search based testability refactoring.**

I. Introduction

Testability transformation produces a version of a program that is more amenable to test data generation [36], [37], [30]. Test data is generated from the transformed version of the program, but it is applied to the original version for testing purposes. A transformation is said to be a *testability transformation* with respect to a test adequacy criterion if all test suites that are adequate for the transformed program are also adequate for the untransformed program.

Testability transformation has proved effective for formal models [15], state based models [51], [54] and MATLAB simulink models [23], as well as for programs in Object Oriented languages [3], and for imperative language testing in the presence of gotos [48], flag variables [5], [11], high–precision arithmetic operations [61] and nesting [59]. Therefore, we can expect that much will be gained by exploiting relationships between testability transformation and refactoring.

Using testability transformation, the transformed program's only role is to help test data generation. However, by construction, the transformed program's set of adequate set suites are also adequate for the original; if a test suite is adequate for the transformed program then it is adequate for the original. The transformed program produced by testability transformation can be discarded once test data has been generated using it, because it will no longer be needed. A testability transformed program is thus merely a means to an end, rather than an end in itself.

This is a subtle shift of emphasis from both traditional program transformation and refactoring. The refactored version becomes the new version of the program once it is accepted by the developer. Therefore, testability transformation may be thought of as a 'speculative refactoring', which does not need to be useful to the developer so long as it allows test data to be generated more easily than can be achieved with the original program.

[1]The paper was written to accompany the author's keynote presentation at REFTEST: The 1st REFactoring and TESTing Workshop, Berlin, March 2011.

When we use refactoring for testing we make the program more amenable to testing. As such, refactoring for testing is akin to testability transformation. However, refactoring a program has the intention that the refactored version of the code becomes the new version of the program, whereas testability transformation typically discards the refactored after test data generation. Once test data has been generated from the testability transformed program, the transformed version is no longer required, because the test data is applied to the original. This has the advantage that the developer need not have to be aware of the transformation process, but need only use the ultimate result; the test data for the original program generated.

However, in refactoring for testing, the goal is to find a version of the program that is a *bone fide* refactoring. That is, one that can satisfactorily replace the original program as the new (and better version of the program). The refactored program must therefore be suitable, not only for testing, but also for all the other activities, centred on subsequent program evolution by the developers and maintainers of the system. For this type of refactoring, the term '*Testability Refactoring*' is introduced in this paper. A testability refactoring meets two objectives simultaneously: it is better for testing and also (at least no worse) for on-going human development and maintenance.

A testability refactoring is a special subclass of testability transformations in which the transformed program is not only useful for test data generation, but is also acceptable to the developer *and* the tester as a better version of the program with which to continue development from the point of refactoring onwards. This paper explores the interplay between these two possibilities.

The rest of this paper is organised as follows: Section II presents a brief overview of the definitions of testability transformation, while Section III introduces the notion of testability refactorings as a subclass of testability transformations. Section IV presents several open problems and directions for future work in the study of testability refactoring, introducing the concepts of test–carrying code, schizophrenic refactoring, search based testability refactoring and human–in–the–loop testability refactoring. Section V concludes.

II. Testability Transformation

To make the paper self-contained, this section revisits the formal definition of a testability transformation [37]. The definition is straightforward, but it opens up some surprising theoretical and practical possibilities, because a testability transformation need not preserve the traditional semantics of the program from which it is constructed.

While testability transformation can preserve traditional semantics *and* improve testability, it is also possible to consider more relaxed classes of testability transformation that merely preserve the semantics captured by the test adequacy criterion. This is a significant departure from traditional program transformation [8], [19], [74].

A *test adequacy criterion* is any set of program elements to be covered during testing. For example, a test adequacy criterion could be a set of Control Flow Graph nodes, branches or paths (at the code level) or a set of use cases, requirements or states (at higher levels of abstraction). Given a test adequacy criterion, a *testing-oriented transformation* is a partial function that maps a program and test adequacy criteria to an updated program and updated test adequacy criteria.

Definition 1 (Testing-Oriented Transformation):
Let **P** be a set of programs and **C** be a set of test adequacy criteria. A program transformation is a partial function in **P** $\rightarrow$ **P**. A *Testing-Oriented Transformation* is a partial function in (**P** $\times$ **C**) $\rightarrow$ (**P** $\times$ **C**).

A testability transformation is merely a special case of a testing-oriented transformation that has properties that make the transformed program suitable for testing the adequacy of the original.

Definition 2 (Testability Transformation):
A Testing-Oriented Transformation, τ is a *Testability Transformation* iff for all programs p, and criteria c, if $\tau(p, c) = (p', c')$ then for all test sets T, T is adequate for p according to c if T is adequate for p' according to c'.

For some criterion, c, a c–preserving testability transformation guarantees that the transformed program is suitable for testing with respect to the original criterion.

Definition 3 (c–Preserving Testability Transformation):
Let τ be a testability transformation. If, for some criterion, c, for all programs p, there exists some program p' such that $\tau(p, c) = (p', c)$, then τ is called a c–preserving testability transformation.

Much of the previous work on testability transformation has used the branch coverage criterion [5], [11], [61], [59]. In this previous work, the testability transformations of interest are those which are '<branch adequacy>–preserving testability transformations' according to Definition 3.

III. Testability Refactoring

Testability transformation can be very different from traditional transformation and also from refactoring, because it need not preserve the traditional semantics of the program to which it is applied. Indeed, the traditional semantics is of little consequence; it only has any influence on the transformations permissible through its influence on the set of adequate test suites that must be preserved by the testability transformation.

As such, a testability transformation does not preserve the traditional semantics, so much as it preserves a new form of semantics defined by the test adequacy criterion. Furthermore, there is not one single semantics preserved by testability transformation. Rather, there is a lattice of related semantics that describe the lattice of associated test adequacy criteria.

For instance, the program $[\![$`if (x>y); else;`$]\!]$ Can be transformed to $[\![$`skip;`$]\!]$ according to traditional semantics. This is one of the elementary transformations that one would expect to be available in almost any traditional transformation toolbox. However, according to the semantics preserved by <branch adequacy>–preserving testability transformation, it *cannot* be transformed to $[\![$`skip`$]\!]$ because this would change the set of branch adequate test suites.

By contrast, the program $[\![$`if (x>y) x=1; else x=2;`$]\!]$ transforms to $[\![$`if (x>y); else;`$]\!]$ according to <branch adequacy>–preserving semantics, but this definitely would not be a transformation that one would want to perform if one were concerned to preserve traditional semantics; the assignments to the variable `x` are lost in this transformation.

There are examples, where transformations such as this, which may appear 'bizarre' traditionally speaking, can be useful because they improve testability. For instance, suppose there is some large part of the program that does not affect whether or not a program covers a branch of interest, b. Suppose further, that b is a particularly difficult branch to cover and a lot of test effort is put in to automatically generate test data to cover it. The process of repeatedly executing the program to attempt to cover b is unnecessarily slowed by the repeated execution of the code which cannot affect b. Testability can therefore be improved (that is, speeded up) by simply removing the code that does not affect branch b.

Clearly, removing a large portion of code will not preserve traditional semantics, but it is acceptable for testability transformation so long as it preserves 'b–coverage adequacy'. This approach has been used to slice away irrelevant parts of the program and its input space in previous work on testability transformation [33], [40], [54], with results that indicate that it can improve both the efficiency and effectiveness of testing.

In program refactoring, the goal of the refactoring process is to produce a better version of the program that is more amenable to the human developer [62]. Refactorings typically aim to make the process of ongoing development more reliable and cheaper. As a result refactoring work tends to focus on the way in which refactoring improves the program for the human reader [18]. This is the principal point of divergence between testability transformation and refactoring. Using testability transformation, the result is intended to be more amenable to testing, while for refactoring it is intended to be more amenable to a human.

Despite their different intentions, testability transformation and refactoring do share an intersection: The two approaches overlap and agree on those sets of transformations that create a new version of the program that is amendable to both the developer and the tester. I propose to use the term *testability refactoring* for this special subclass of testability transformations. Its definition is simple:

Definition 4 (Testability Refactoring):
A *testability refactoring* is any refactoring that is also a testability transformation.

IV. Directions for Testability Refactoring

This section introduces four ways in which Testability Refactoring might be achieved: Test–Carrying Code, Schizophrenic Refactoring, Search Based Testability Refactoring and Human–in–the–loop Testability Refactoring.

A. *Test-carrying code*

The idea of proof-carrying code is familiar in the field of program verification [63]. However, there appears to be no equivalent concept of *test–carrying code* in the world of software testing. Such a concept is overdue. One could envisage a testability refactoring approach in which the test cases to be applied to a program are bound together with the program code. Naturally, the development environment would have to offer the ability to switch on and off this additional 'view' of the software; its test oracle.

In some ways, the idea of test–carrying code resembles Knuth's proposal of literate programming [52]. Using literate programming the idea is that the program should include, not merely comments, but the entire documentation of the program, written as an explanation to another programmer. In fact, for a literate program, the idea is that the *primary* task of a programmer is to explain to another programmer what the program is intended to do in prose (the literate part) and to accompany this prose with the code that implements it.

To make the approach practical, Knuth proposed a system in which the literate program could be projected, using $\TeX$, into a `dvi` file, which rendered the prose description of the program as a `dvi` document for human consumption, along with a separate projection, which extracts the code for the machine. Updates are performed on code and prose together so that the code and its description remain consistent.

Knuth introduces the idea of literature programming in 1984. It has proved influential, though it has not been widely taken up directly as Knuth proposed. Echoes of this approach and its underlying motivation can be found in more recent work on pair programming [79] and model view consistent updates issues [25] as well as in typical development environments such as JavaDoc [55].

The key observation behind literate programming is that programs are, or should be, written for several 'audiences'; humans and machines. The notion of test–carrying code shares this core motivation. However, rather than focusing on a documentation of a program in prose, I prefer to consider the best form of 'documentation' to be a set of well–chosen and well–explained test cases.

One of the biggest problems with testing is the lack of an automated oracle [10] and the costs of human generation of these oracles when all else fails [42]. However, there are many techniques for software test data generation, such as Search Based Software Testing [2], [27], [60] and Dynamic Symbolic Execution [24], [70] that address at least half of this problem. That is, these techniques automate the process of generating test inputs, though they do not assist with the determination of a suitable outputs to accompany the corresponding inputs they generate. This is the oracle problem: what is a correct output for a given input.

Notwithstanding the oracle problem, there are many situations in which there may be some form of 'partial oracle'. For instance, when the project has some agreed set of test cases derived from requirements [58] or specifications [49], or when they can be inferred from other documentation [66]. Also, when regression testing [82] or when using metamorphic testing [17], there is a partial oracle provided, either by the regression test suite or the metamorphic relations. In all of these situations it makes sense for the test suites to reside together with the code they test and for the mapping between tester and tested to be maintained.

It is in the maintenance of this mapping between code and the test carried with it, that testability refactoring has a role to play. As the code is refactored, the test carried with the code must also be refactored in a manner that preserves the consistency of the mapping. For example, when a refactoring moves a method from one class to another, then the method–carried test code should migrate with the code. This is the meaning I wish to ascribe to the concept of 'test–carrying code'. It would seem to be a natural way to maintain the resources and effort bound up in the test cases of the overall system.

Also, given that the oracle problem is so pressing and programs often have inadequate test suites on which we can rely, is it not all the more important that we should do all we can to preserve those few good quality test cases we do have? Testability refactoring can achieve this by ensuring that the test suite carried by a code fragment, really is carried along with the fragment as it migrates and mutates under development, evolution, refactoring and transformation. In this way we preserve the relationship between our code and its test suite, helping to preserve the oracle information we have available to us.

B. *The Schizophrenic Refactoring*

Testability transformation seeks to make the code easier to test. This has traditionally been applied to automated test data generation [3], [11], [23], [61]. For such applications, the code produced by testability transformation need only be readable by a machine. It is well known that machine–generated code is not human-friendly [47], [80] and so this form of testability transformation is not well suited, as it stands, to testability refactoring.

For testability refactoring we seek a transformation that is useful to human and machine. This may not always be achievable, but it should not be a Boolean goal, which is either satisfied or unsatisfied. It is more nuanced than that. We can measure the degree to which our refactoring serves the human and the degree to which it serves the machine and seek to obtain as much as possible of both. The term '*Schizophrenic Refactoring*' will be used to refer to a refactoring that tries to achieve two (possibly conflicting) goals, seeking to balance each against one another, where both cannot be simultaneously achieved. Sometimes we may be lucky: the two objectives may not even be in opposition. There are testability transformation approaches, though they are aimed at improving testability, which may also improve human readability. This section explores three such examples, concerned with transformations to removal flags, goto statements and side effects, each of which is arguably bad for both test generation and for comprehension.

1) Flag Removal as a Testability Refactoring: Testability transformation has been used to replace a flag with a more expressive arithmetic computation that captures the meaning of the flag at the point at which it is used. Consider the illustrative example in Figure 1. The original code fragment is shown in Figure 1(a), in which the code denoted '...' is some arbitrary code segment that does not mention the flag variable. It has been shown that search based testing has difficulties covering branches like the last one in this fragment because of the presence of flag variables which create a so-called 'needle–in–a–haystack" fitness landscape [37]. Many authors have proposed solutions to this problem [6], [56], [57], [75], the first of which was to use a testability transformation [36]. The testability transformation which produces the transformed version in Figure 1(b) has been implemented in a tool called FlagRemover [11], which is now publicly available.

The transformed version in Figure 1(b) replaces the reference to a flag by an expression that is much longer, but which draws together the expressions that denote the outcome of the flag as a predicate. In transforming the program in this way, the testability transformation also transforms the fitness landscape, so that the 'haystack' becomes a single, smooth, rounded 'hill of hay', in which the 'needle' is conveniently located at its summit. The 'needle' in this analogy is the desired input vector that causes the flag to evaluate to true. Local search, such as hill climbing algorithms are fast and efficient at traversing such simple 'hill–like' landscapes [44]. The testability transformation therefore makes the problem of covering the flag-controlled branch much more amenable to search based testing.

It could be argued that the version of the fragment in Figure 1(b) is also more easy to understand *at the point of the final use of the variable* flag. The testability transformation has gathered together all the relevant code that expresses the meaning of flag at the point at which the flag is used. Though this leads to a longer expression, it saves the human reader from having to search to locate the relevant computations and thereby may reduce cognitive effort.

There are no empirical studies of the effect of this flag removal transformation on human comprehension of the transformed code and so any claim about the human-readability of flag-removal testability transformation remains anecdotal at this stage. An empirical study of flag removal for comprehension would answer the question 'Is flag removal testability transformation also a testability refactoring?'.

2) goto Removal as a Testability Refactoring: Unstructured or 'goto' code has also been shown to be a problem for test data generation techniques for which a testability transformation can be used to address the problem [48]. Of course, the debate about whether goto statements are harmful to program comprehension dates back to Dijkstra's famous publication in Communication of the ACM [20] and was the subject of much debate in the literature throughout the following decades.

`flag = n<4;`	
`...`	`...`
`if (n%2==0) flag=0;`	`n' = n;`
	`flag=(n'%2==0)?0:(n'<4);`
`...`	`...`
`if (a[i]!='0'&&flag)`	`if (a[i]!='0' &&`
	`    (n'%2==0)?0:(n'<4))`
`   ...`	`   ...`
(a) Original	(b) Transformed version

Fig. 1. Flag Removal Example [36]. The transformed version enriches the expression that captures the value of the flag at its final use. This has been shown to be of benefit to automated test data generation. It may also be useful for certain comprehension activities.

Böhm and G. Jacopini [12] had demonstrated that goto statements were not essential, in the sense that a program could be transformed to a goto–free version with equivalent behaviour. Ashcroft and Manna realised this theoretical observation in the first algorithm for transforming unstructured programs to versions with no goto statements [4], while Hopcroft, in a contribution to a paper by Knuth and Floyd [53], demonstrated that unstructured programs could not always be transformed to goto–free versions under path equivalence. That is, though we may transform a goto–program into a goto–free version that performs the same computation, it may have to achieve the same results by following different paths. Clearly, this may affect the comprehension of the code, particularly if the programmer was originally familiar with the structure and behaviour of the original.

The contortions sometimes required to remove goto statements from code in which they were heavily used, promoted some to reject goto–removal transformation [69]. The debate continued well into the 1980s, spawning some innovative paper titles [77], which played upon the title of Dijkstra's original publication[2]. It is interesting to note that, despite these many papers, often authored by Turing–award–winning authors, with interesting and important theoretical insights into the interplay between program structure and behaviour, there remains *no* thorough empirical study of the effect of the goto statement on program comprehension.

Empirical studies have been performed on program corpuses in order to empirically investigate the ways in which goto statements are used in code [9], but there appears to be no psychological empirical study of programmers themselves to determine, empirically, the degree to which goto statements may affect program comprehension. It therefore remains open as to whether a goto–removal testability transformation is also a testability refactoring, though anecdotally, we may suspect that it is.

This lack of empirical evidence is surprising, given that the 'goto controversy' is so well established in the folklore of computer science, while more recent trends such as object orientation have been more thoroughly studied, providing a

[2]Actually, Dijkstra's famous 'paper' on goto statements was nothing of the kind; it was, in fact, a short letter to the editor of CACM, in which Dijkstra made the anecdotal observation that better programmers refrain from using goto statements. The editor gave the letter its title, simply putting Dijkstra's sentiment into the passive voice and so the title given to the letter became 'goto considered harmful'.

wealth of empirical evidence [14], [16], [26], as has the topic of refactoring itself [1], [64], [73].

3) Side Effect Removal as a Testability Refactoring: Side effects are also a problem for test data generation techniques [7], [60]. Side effect-removal transformation can also act as a testability transformaton to assist with this problem. Side effect removal has *also* been suggested as an aid to program comprehension [39]. For example, using post placement side effect removal transformation [38], the code fragment

⟦`if (++i && i--) x=1;`⟧

is transformed to

⟦`if (i==-1) i = i+1; else x = 1;`⟧

The first version of the code fragment is hard to understand because the side effects are mingled with the difficulty of determining the effects of predicate evaluation in the presence of short–circuit evaluation. In code with side effects, expressions play two roles, one of which is evaluation, while the other is state update. This makes comprehension hard because the reader has to keep a mental track of the state as the expression is evaluated. The presence of side effects also inhibits understanding because it destroys the mathematical interpretation of the expressions. The desire to avoid side effects in order to retain these mathematical properties was one of the motivations for functional programming [50].

In the transformed, side–effect–free version of the code fragment, expressions serve only one purpose: evaluation, and their mathematical character is restored to them, imbuing them with many of the properties that their mathematical counterparts enjoy. All state update is performed in statements, where the sequence of execution steps is comparatively easier to follow. This has a positive effect on program comprehension.

There are empirical results which have provided evidence that programmers, whether they be novices and experienced coders, perform better at cognitive tasks when faced with side–effect–free code fragments than with their side–effecting equivalents [21]. As such, side effect removal transformation can be said to be a testability refactoring; it makes code better for comprehension and testing.

C. Search Based Testability Refactoring

Search based Software Engineering (SBSE) [2], [27], [28], [41], [68] has been widely used as a way to find good refactorings from the large space of potential transformations that can be applied [13], [45], [64], [65], [71].

Using Search Based Refactoring, the goal is to define the metrics that capture properties of programs that we seek to improve through refactoring. The metrics are re-formulated as fitness functions [31]. The fitness functions are used to guide a search based optimisation algorithm, such as a hill climber or a genetic algorithm. The fitness captures the metric of interest so that when the search moves to a better solution according to the guidance of the fitness function, that solution should denote a better program according to the metric of interest. In this way the search gradually moves the program towards a refactored version which optimises the value of the metric of interest.

Typically there may be very many metrics to consider in refactoring [46], all of which have a bearing on the best sequence of refactorings to perform. In previous work this multiplicity of metrics has been handled, either by combining a set of metrics into a single weighted sum [65], [71] or by adopting a pareto optimal approach [45].

The weighting approach is suitable when one can, *a priori*, determine the relative importance of each of the metrics to be combined by weighted sum. Where such a weighing scheme is not obvious a pareto optimal approach is more suitable [28], [45].

Using Pareto optimality, it is not possible to measure 'how much' better one solution is than another, merely to determine whether one solution is better than another. Suppose F denotes the combined fitness of a set of individual fitness functions f_i each of which operate on a vector of values $\overline{x}$. A solution $\overline{x_1}$ is considered to be a (pareto) improvement on $\overline{x_2}$, if and only if

$$\forall i . f_i(\overline{x_1}) \geq f_i(\overline{x_2}) \quad \wedge \quad \exists i . f_i(\overline{x_1}) > f_i(\overline{x_2})$$

That is, $\overline{x_1}$ is better than $\overline{x_2}$ if and only if it is better according to at least one of the individual fitness functions and no worse according to all of them. In this situation $\overline{x_1}$ is said to 'dominate' $\overline{x_2}$.

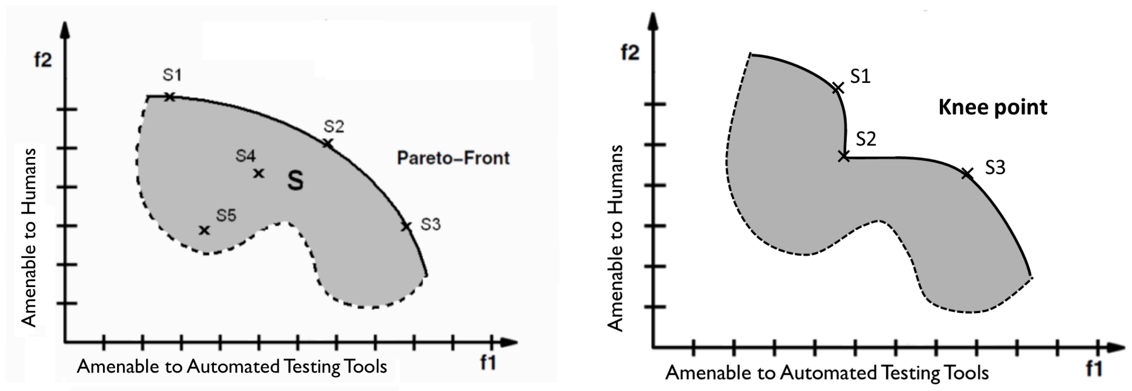
Using Pareto optimality, the search process yields, not a single optimised solution, but a set of solutions that are non-dominated. That is, each member of the non-dominated set is no worse than any of the others in the set, but also cannot be said to be better. Any such set of non-dominated solutions forms a Pareto front.

For the problem of balancing the two objectives of refactoring (testing and comprehension) we have just such a multi objective scenario in which we have two different and potentially conflicting objectives: refactor for human and refactor for machine. These objectives may be in conflict and it is not obvious how to choose to weight their importance. As such, this problem is very well suited to pareto optimal search based refactoring. This is illustrated in Figure 2.

In Figure 2(a) the pareto front is smooth and unbroken. This is an idealised scenario. The front may be broken, because some solutions may be infeasible. The front can indicate this, guiding the decision maker. Furthermore, many SBSE pareto fronts (for example those used in regression testing [81] and requirements optimization [83]) have been found to exhibit such 'knee points' at which the trade off between the two objectives changes dramatically. In the example in Figure 2(b), the decision maker would be likely to choose solution either $S1$ or $S3$ but not $S2$ even though all are non–dominated. This is because the pareto front reveals that solution $S2$ dramatically reduces achievement of one of the two objectives of with little impact on the other, when compared to either $S1$ or $S3$.

D. Human–in–the–loop Testability Refactoring

Testability transformation has previously been targeted at situations in which automated techniques have difficulty in generating test data. This difficulty is due to some structural



(a) Smooth Pareto Front (b) Pareto Front with Knee point

Fig. 2. Pareto fronts, illustrating a potential trade off between human– and machine– friendly transformation in the search for a testability refactoring. Fitness function $f1$ measures how suitable the refactored program is for humans, while fitness function $f2$ measures how suitable it is for automated testing. Solutions $S1$, $S2$ and $S3$ each denote different tradeoff between $f1$ and $f2$. None can be said to be better, because each lies on the pareto front of non–dominated solutions. Computing this pareto front provides decision support.

feature of the code for which the approach (or a tool that improves it) proves to be ineffective or inefficient. However, there is no reason why a testability transformation could not be targeted at the human tester, who retains some test objective for which no tool can help, even after some tool-oriented testability transformation has been performed.

For example, suppose that a program contains a branch that no tool has been able to cover. It could be that this branch is infeasible, in which case attempts to cover it will be futile. However, statement reachability (and hence branch coverage) is known to be undecidable [78] so we cannot expect a machine to be able to reliably answer this question for us. As a result, it is likely that a human may be faced with this time–consuming and often tedious task.

This is not an unrealistic scenario: Standards for testing, such as the Aerospace Standard [67] lay down the requirement that 100% feasible branch coverage (and higher) levels of test adequacy criteria must be achieved. In order to meet these requirements, a tester may employ tools, but the undecidability of branch coverage inevitably means that there may remain uncovered branches for which the human has either to find a test case or to determine that the branch is uncoverable.

In such situations, though testability transformation may not assist directly by helping a tool to find a test case, it may still assist indirectly, by helping the human to address the problem faster. Program analysis and manipulation has been previously suggested as a way to help a human analyst to find answers to undecidable propositions [32] and this approach could also be used to help the human solve the ‘uncovered branch problem’.

For example, slicing [35], [72], [76] could be used to remove parts of the program that cannot affect the branch of interest, thereby simplifying the task. Furthermore, in those cases where the programmer is aware of pre– and post–conditions under which the code must be executed, pre/post conditioned slicing [34] can be used to further refine the slice, narrowing attention on the parts of the code that must be considered in order to determine whether the branch is feasible or not. In this application the slicing algorithms would be playing the role of testability refactoring.

Another possible way of keeping the human in the loop during the refactoring process, would be to use interactive evolution [22] for a search based approach to meet comprehension goals. The use of interactive evolution for search based comprehension has previously been advocated [29] and could be used in this context to achieve the human oriented fitness assessment outlined in Section IV-C.

V. CONCLUSION

Testability transformations change programs to make them better suited to automated test data generation. Refactoring, on the whole, changes programs to make them better suited to on-going human activities. I argue that at testability refactoring should seek to do both simultaneously.

This paper presented testability transformation examples that are also testability refactorings because they improve code for both human consumption and also for automated test data generation. There are surely other ways in which programs can be transformed to make them easier for testing and for the programmer. This remains a topic for future investigation.

Refactoring for testing, the subject of this workshop, is all about the question of how to refactor a program in such a way that it makes it easier to test. Because refactoring for testing is also, by definition, *refactoring*, it also surely seeks to improve the program for human-based activities. As such, refactoring for testing is ‘testability refactoring’ as I define it. There is much interesting work to be done to determine what of the previous work on testability transformation can be carried over into testability refactoring. There is also much further work to be done on Testability Refactoring itself.

Acknowledgements:
I am grateful to Kelly Androutsopoulos, André Baresel, Dave Binkley, John Clark, Steve Counsell, Sebastian Danicic, Chris Fox, Robert Hierons, John Howroyd, Bogdan Korel, Bill Langdon, Kiran Lakhotia, Phil McMinn, Marc Roper, Harman Sthamer, Paolo Tonella, Joachim Wegener and Martin Woodward for many rewarding discusssions on the subject of

Testability Transformation. The pareto front diagrams in Figure 2 were produced by Yuanyuan Zhang and have appeared previously in a different form [43].

References

[1] Deepak Advani, Youssef Hassoun, and Steve Counsell. Extracting refactoring trends from open-source software and a possible solution to the 'related refactoring' conundrum. In Hisham Haddad, editor, *ACM Symposium on Applied Computing (SAC 06)*, pages 1713–1720, Dijon, France, 2006. ACM.

[2] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering*, 2010. To appear.

[3] Andrea Arcuri and Xin Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008.

[4] Edward A. Ashcroft and Zohar Manna. The translation of `goto` programs into `while` programs. In C. V. Freiman, J. E. Griffith, and J. L. Rosenfeld, editors, *Proceedings of IFIP Congress 71*, volume 1, pages 250–255. North-Holland, 1972.

[5] André Baresel, David Wendell Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop–assigned flags: A testability transformation approach. In *International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 108–118, Omni Parker House Hotel, Boston, Massachusetts, July 2004. Appears in Software Engineering Notes, Volume 29, Number 4.

[6] André Baresel and Harmen Sthamer. Evolutionary testing of flag conditions. In *Genetic and Evolutionary Computation (GECCO-2003)*, volume 2724 of *LNCS*, pages 2442–2454, Chicago, 12-16 July 2003. Springer-Verlag.

[7] André Baresel, Harmen Sthamer, and Michael Schmidt. Fitness function design to improve evolutionary structural testing. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1329–1336, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[8] Ira D. Baxter. Transformation systems: Domain-oriented component and implementation knowledge. In *Proceedings of the Ninth Workshop on Institutionalizing Software Reuse*, Austin, TX, USA, January 1999.

[9] Barbara A. Benander, Narasimhaiah Gorla, and Alan C. Benander. An empirical study of the use of the GOTO statement. *The Journal of Systems and Software*, 11(3):217–223, March 1990.

[10] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In Lionel Briand and Alexander Wolf, editors, *Future of Software Engineering 2007*, Los Alamitos, California, USA, 2007. IEEE Computer Society Press.

[11] David Binkley, Mark Harman, and Kiran Lakhotia. FlagRemover: A testability transformation for transforming loop assigned flags. *ACM Transactions on Software Engineering and Methodology*, 2010. to appear.

[12] C. Böhm and G. Jacopini. Flow diagrams, turing machines, and languages with only two formation rules. *Communications of the ACM*, 9(5):366–372, May 1966.

[13] Salah Bouktif, Giuliano Antoniol, Ettore Merlo, and Markus Neteler. A novel approach to optimize clone refactoring activity. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1885–1892, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[14] Lionel C. Briand, Erik Arisholm, Steve Counsell, Frank Houdek, and Pascale Thévenod-Fosse. Empirical studies of object-oriented artifacts, methods, and processes: State of the art and future directions. *Empirical Software Engineering*, 4(4):387–404, 1999.

[15] Achim D. Brucker, Lukas Brügger, Paul Kearney, and Burkhart Wolff. Verified firewall policy transformations for test case generation. In $3^{rd.}$ *International Conference on Software Testing, Verification and Validation (ICST 2010)*, pages 345–354. IEEE Computer Society, 2010.

[16] Michelle Cartwright and Martin J. Shepperd. An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Engineering*, 26(8):786–796, 2000.

[17] Tsong Yueh Chen, Jianqiang Feng, and T. H. Tse. Metamorphic testing of programs on partial differential equations: A case study. In 26^{th} *Annual International Computer Software and Applications Conference (COMPSAC'02)*, pages 327–333. IEEE Computer Society, 2002.

[18] A. Correa, C. Werner, and M. Barros. Refactoring to improve the understandability of specifications written in object constraint language. *IET Software*, 3(2):69–90, 2009.

[19] John Darlington and Rod M. Burstall. A tranformation system for developing recursive programs. *Journal of the Association for Computer Machinery*, 24(1):44–67, 1977.

[20] Edskar Wabe Dijkstra. Goto statement considered harmful. *Communications of the ACM*, 11:147, 1968.

[21] José Javier Dolado, Mark Harman, Mari Carmen Otero, and Lin Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Transactions on Software Engineering*, 29(7):665–670, 2003.

[22] Pablo Funes, Eric Bonabeau, Jerome Herve, and Yves Morieux. Interactive multi-participant task allocation. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 1699–1705, Portland, Oregon, 20-23 June 2004. IEEE Press.

[23] Kamran Ghani and John A. Clark. Widening the goal posts: Program stretching to aid search based software testing. In *Proceedings of the 1st International Symposium on Search Based Software Engineering (SSBSE'09)*, Windsor, UK, 13-15 May 2009. IEEE.

[24] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223. ACM, 2005.

[25] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. 6th International Conference on Model Languages and Applications*, volume 2863 of *LNCS*, pages 144–158, San Francisco, CA, USA, 2003. Springer.

[26] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

[27] Mark Harman. Automated test data generation using search based software engineering. In 2^{nd} *International Workshop on Automation of Software Test (AST 07)*, page 2, Minneapolis, USA, May 2007. IEEE Computer Society Press.

[28] Mark Harman. The current state and future of search based software engineering. In Lionel Briand and Alexander Wolf, editors, *Future of Software Engineering 2007*, pages 342–357, Los Alamitos, California, USA, 2007. IEEE Computer Society Press.

[29] Mark Harman. Search based software engineering for program comprehension. In 15^{th} *International Conference on Program Comprehension (ICPC 07)*, pages 3–13, Banff, Canada, 2007. IEEE Computer Society Press.

[30] Mark Harman. Open problems in testability transformation. In *1st International Workshop on Search Based Testing (SBT 2008)*, Lillehammer, Norway, 2008. Keynote paper.

[31] Mark Harman and John Clark. Metrics are fitness functions too. In 10^{th} *International Software Metrics Symposium (METRICS 2004)*, pages 58–69, Los Alamitos, California, USA, September 2004. IEEE Computer Society Press.

[32] Mark Harman, Chris Fox, Rob Mark Hierons, David Wendell Binkley, and Sebastian Danicic. Program simplification as a means of approximating undecidable propositions. In 7^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'99)*, pages 208–217, Los Alamitos, California, USA, May 1999. IEEE Computer Society Press.

[33] Mark Harman, Youssef Hassoun, Kiran Lakhotia, Philip McMinn, and Joachim Wegener. The impact of input domain reduction on search-based test data generation. In *ACM Symposium on the Foundations of Software Engineering (FSE '07)*, pages 155–164, Dubrovnik, Croatia, September 2007. Association for Computer Machinery.

[34] Mark Harman, Rob Mark Hierons, Sebastian Danicic, John Howroyd, and Chris Fox. Pre/post conditioned slicing. In *IEEE International Conference on Software Maintenance (ICSM'01)*, pages 138–147, Los Alamitos, California, USA, November 2001. IEEE Computer Society Press.

[35] Mark Harman and Robert Mark Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[36] Mark Harman, Lin Hu, Robert Hierons, André Baresel, and Harmen Sthamer. Improving evolutionary testing by flag removal. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1359–1366, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[37] Mark Harman, Lin Hu, Robert Mark Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, January 2004.

[38] Mark Harman, Lin Hu, Robert Mark Hierons, Xingyuan Zhang, Malcolm Munro, José Javier Dolado, Mari Carmen Otero, and Joachim

Wegener. A post-placement side-effect removal algorithm. In *IEEE International Conference on Software Maintenance*, pages 2–11, Los Alamitos, California, USA, October 2002. IEEE Computer Society Press.

[39] Mark Harman, Lin Hu, Xingyuan Zhang, and Malcolm Munro. Side-effect removal transformation. In 9th *IEEE International Workshop on Program Comprehension*, pages 310–319, Los Alamitos, California, USA, May 2001. IEEE Computer Society Press.

[40] Mark Harman, Fayezin Islam, Tao Xie, and Stefan Wappler. Automated test data generation for aspect-oriented programs. In 8th *International Conference on Aspect-Oriented Software Development (AOSD '09)*, pages 185–196, Charlottesville, Virginia, USA, March 2009.

[41] Mark Harman and Bryan F. Jones. Search based software engineering. *Information and Software Technology*, 43(14):833–839, December 2001.

[42] Mark Harman, Sung Gon Kim, Kiran Lakhotia, Philip McMinn, and Shin Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In 3rd *International Workshop on Search-Based Software Testing (SBST 2010)*, Paris, France, April 2010.

[43] Mark Harman, Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, Department of Computer Science, King's College London, April 2009.

[44] Mark Harman and Philip McMinn. A theoretical and empirical study of search based testing: Local, global and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[45] Mark Harman and Laurence Tratt. Pareto optimal search-based refactoring at the design level. In *GECCO 2007: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1106 – 1113, London, UK, July 2007. ACM Press.

[46] Rachel Harrison, Steve Counsell, and Reuben V. Nithi. An investigation into the applicability and validity of object-oriented design metrics. *Empirical Software Engineering*, 3(3):255–273, 1998.

[47] Mats Per Erik Heimdahl and David J. Keenan. Generating code from hierarchical state-based requirements. In *Proceedings: 3rd IEEE International Symposium on Requirements Engineering*, pages 210–221. IEEE Computer Society Press, 1997.

[48] Robert Hierons, Mark Harman, and Chris Fox. Branch-coverage testability transformation for unstructured programs. *The computer Journal*, 48(4):421–436, 2005.

[49] Robert M. Hierons. Testing from a Z specification. *Software Testing, Verification and Reliability*, 7(1):19–33, 1997.

[50] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Principles of Programming Languages (POPL'93)*, pages 71–84, 1993.

[51] AbdulSalam Kalaji, Robert Mark Hierons, and Stephen Swift. A testability transformation approach for state-based programs. In ^1st *International Symposium on Search Based Software Engineering (SSBSE 2009)*, pages 85–88, Windsor, UK, May 2009. IEEE.

[52] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[53] Donald E. Knuth and Robert W. Floyd. Notes on avoiding "go to" statements. *Information Processing Letters*, 1(1):23–31, February 1971.

[54] Bogdan Korel, Mark Harman, S. Chung, P. Apirukvorapinit, and R. Gupta. Data dependence based testability transformation in automated test generation. In 16th *International Symposium on Software Reliability Engineering (ISSRE 05)*, pages 245–254, Chicago, Illinios, USA, November 2005.

[55] Douglas Kramer. API documentation from source code comments: A case study of Javadoc. In *Proceedings of the 7th Annual International Conference of Computer Documentation (SIGDOC-99)*, pages 147–153, N.Y., September 12–14 1999. ACM Press.

[56] Xiyang Liu, Ning Lei, Hehui Liu, and Bin Wang. Evolutionary testing of unstructured programs in the presence of flag problems. In 12th *Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 525–533. IEEE Computer Society, 2005.

[57] Xiyang Liu, Hehui Liu, Bin Wang, Ping Chen, and Xiyao Cai. A unified fitness function calculation rule for flag conditions to improve evolutionary testing. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 337–341, Long Beach, CA, USA, 2005. ACM.

[58] Neil A. Maiden and Cornelius Ncube. Acquiring COTS software selection requirements. *IEEE Software*, 15(2):46–56, 1998.

[59] Phil McMinn, David Binkley, and Mark Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering and Methodology*, 18(3), May 2009. Article 11.

[60] Philip McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[61] Philip McMinn. Search-based failure discovery using testability transformations to generate pseudo-oracles. In Franz Rothlauf, editor, *Genetic and Evolutionary Computation Conference (GECCO 2009)*, pages 1689–1696, Montreal, Québec, Canada, 2009. ACM.

[62] Tom Mens and Tom Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.

[63] G. Necula. Proof-carrying code. In *Principles of Programming Languages (POPL'97)*, pages 106–119. ACM Press, 1997.

[64] Mark O'Keeffe and Mel Ó Cinnéide. Search-based refactoring: an empirical study. *Journal of Software Maintenance*, 20(5):345–364, 2008.

[65] Mark O'Keeffe and Mel O'Cinneide. Search-based software maintenance. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 249–260, March 2006.

[66] Dennis K. Peters and David Lodge Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, 1998.

[67] Radio Technical Commission for Aeronautics. RTCA DO178-B Software considerations in airborne systems and equipment certification, 1992.

[68] Outi Räihä. A survey on search–based software design. *Computer Science Review*, 4(4):203–249, 2010.

[69] F. Rubin. "GOTO considered harmful" considered harmful. *Communications of the ACM*, 30(3):195–196, March 1987.

[70] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Michel Wermelinger and Harald Gall, editors, 10th *European Software Engineering Conference and 13th ACM International Symposium on Foundations of Software Engineering (ESEC/FSE '05)*, pages 263–272. ACM, 2005.

[71] Olaf Seng, Johannes Stammel, and David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Genetic and evolutionary computation conference (GECCO 2006)*, volume 2, pages 1909–1916, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[72] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 2011. to appear.

[73] Paolo Tonella and Mariano Ceccato. Refactoring the aspectizable interfaces: An empirical assessment. *IEEE Transactions on Software Engineering*, 31(10):819–832, 2005.

[74] Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.

[75] Stefan Wappler, Joachim Wegener, and André Baresel. Evolutionary testing of software with function-assigned flags. *The Journal of Systems and Software*, 82(11):1767–1779, November 2009.

[76] Mark Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[77] P. H. Welch. GOTO (Considered Harmful)^n, n is Odd. *Occam User Group Newsletter*, 8:22–26, January 1988.

[78] Elaine J. Weyuker. *Program schemas with semantic restrictions*. PhD thesis, Rutgers University, New Brunswick, New Jersey, June 1977.

[79] Laurie A. Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the case for pair programming. *IEEE Software*, 17(4), 2000.

[80] Hui Wu, Jeffrey G. Gray, Suman Roychoudhury, and Marjan Mernik. Weaving a debugging aspect into domain-specific language grammars. In Hisham Haddad, Lorie M. Liebrock, Andrea Omicini, and Roger L. Wainwright, editors, *ACM Symposium on Applied Computing (SAC 2005)*, pages 1370–1374, Santa Fe, New Mexico, USA, March 2005. ACM.

[81] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *International Symposium on Software Testing and Analysis (ISSTA'07)*, pages 140 – 150, London, United Kingdom, July 2007. Association for Computer Machinery.

[82] Shin Yoo and Mark Harman. Regression testing minimisation, selection and prioritisation: A survey. *Journal of Software Testing, Verification and Reliability*, 2011. To appear.

[83] Yuanyuan Zhang, Anthony Finkelstein, and Mark Harman. Search based requirements optimisation: Existing work and challenges. In *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'08)*, volume 5025, pages 88–94, Montpellier, France, 2008. Springer LNCS.